Functions and an Introduction to Recursion Part III

C++ How to Program, Late Objects Version, 7/e

Dr. Jian-Ren Hou

5.19 Function Templates

- Overloaded functions are normally used to perform similar operations that involve different program logic on different data types.
- If the program logic and operations are identical for each data type, overloading may be performed more compactly and conveniently by using function templates.
- You write a single function template definition.
- Given the argument types provided in calls to this function, C++ automatically generates separate function template specializations to handle each type of call appropriately.
- Thus, defining a single function template essentially defines a whole family of overloaded functions.

5.19 Function Templates (cont.)

- Figure 5.25 contains the definition of a function template (lines 3–17) for a maximum function that determines the largest of three values.
- Function template definitions begin with template (line 3) followed by a template parameter list enclosed in angle brackets (< and >).
- Every parameter in the template parameter list (often referred to as a formal type parameter) is preceded by keyword typename or keyword class (which are synonyms in this context).
- The formal type parameters are placeholders for fundamental types or user-defined types.
- These placeholders are used to specify the types of the function's parameters (line 4), to specify the function's return type (line 4) and to declare variables within the body of the function definition (line 6).
- A function template is defined like any other function, but uses the formal type parameters as placeholders for actual data types.

```
1  // Fig. 5.25: maximum.h
2  // Definition of function template maximum.
3  template < class T >  // or template< typename T >
4  T maximum( T value1, T value2, T value3 )
5  {
6     T maximumValue = value1; // assume value1 is maximum
7
8     // determine whether value2 is greater than maximumValue
9     if ( value2 > maximumValue )
10       maximumValue = value2;
11
12    // determine whether value3 is greater than maximumValue
13    if ( value3 > maximumValue )
14       maximumValue = value3;
15
16    return maximumValue;
17 } // end function template maximum
```

Fig. 5.25 | Function template `maximum` header file.

5.19 Function Templates (cont.)

- The function template in Fig. 5.25 declares a single formal type parameter T (line 3) as a placeholder for the type of the data to be tested by function maximum.
- The name of a type parameter must be unique in the template parameter list for a particular template definition.
- When the compiler detects a maximum invocation in the program source code, the type of the data passed to maximum is substituted for T throughout the template definition, and C++ creates a complete function for determining the maximum of three values of the specified data type—all three must have the same type, since we use only one type parameter in this example.
- Then the newly created function is compiled.
- Thus, templates are a means of code generation.
- Figure 5.26 uses the maximum function template (lines 17, 27 and 37) to determine the largest of three int values, three double values and three char values, respectively.

```
1  // Fig. 5.26: fig05_26.cpp
2  // Function template maximum test program.
3  #include <iostream>
4  #include "maximum.h" // include definition of function template maximum
5  using namespace std;
6
7  int main()
8  {
9     // demonstrate maximum with int values
10    int int1, int2, int3;
11
12    cout << "Input three integer values: ";
13    cin >> int1 >> int2 >> int3;
14
15    // invoke int version of maximum
16    cout << "The maximum integer value is: "
17       << maximum( int1, int2, int3 );
18
19    // demonstrate maximum with double values
20    double double1, double2, double3;
21
22    cout << "\n\nInput three double values: ";
23    cin >> double1 >> double2 >> double3;
24
```

Fig. 5.26 | Demonstrating function template `maximum`. (Part 1 of 2.)

```
25      // invoke double version of maximum
26      cout << "The maximum double value is: "
27         << maximum( double1, double2, double3 );
28
29      // demonstrate maximum with char values
30      char char1, char2, char3;
31
32      cout << "\n\nInput three characters: ";
33      cin >> char1 >> char2 >> char3;
34
35      // invoke char version of maximum
36      cout << "The maximum character value is: "
37         << maximum( char1, char2, char3 ) << endl;
38   } // end main
```

```
Input three integer values: 1 2 3
The maximum integer value is: 3

Input three double values: 3.3 2.2 1.1
The maximum double value is: 3.3

Input three characters: A C B
The maximum character value is: C
```

Fig. 5.26 | Demonstrating function template `maximum`. (Part 2 of 2.)

In-Class Exercise

- Apply template to create a simple function myAdd(), which is able to handle the addition of two inputted arguments disregarding their type.

Unfinished function:

```
myAdd(x, y){
        return x+y;
}
```

Please copy and paste the follows in your main function and see what is printed.

```
cout<< myAdd(1,3) << endl;
cout<< myAdd(1.2,3.2) <<endl;
cout<< myAdd('0','1') <<endl;
```

5.20 Recursion

- The programs we've discussed are generally structured as functions that call one another in a disciplined, hierarchical manner.
- For some problems, it's useful to ***have functions call themselves***.
- A recursive function is a function that calls itself, either directly, or indirectly (through another function).
- Recursion is an important topic discussed at length in upper-level computer science courses.
- This section and the next present simple examples of recursion.

5.20 Recursion (cont.)

- Recursive problem-solving approaches have a number of elements in common.
- A recursive function is called to solve a problem.
- The function actually knows how to solve only the simplest case(s), or so-called ***base case(s)***.
- If the function is called with a base case, the function simply returns a result.
- If the function is called with a more complex problem, it typically divides the problem into two conceptual pieces—a piece that the function knows how to do and a piece that it does not know how to do.
- To make recursion feasible, the latter piece must resemble the original problem, but be a slightly simpler or smaller version.
- This new problem looks like the original, so the function calls a copy of itself to work on the smaller problem—this is referred to as a recursive call and is also called the recursion step.

5.20 Recursion (cont.)

- The recursion step often includes the keyword return, because its result will be combined with the portion of the problem the function knew how to solve to form the result passed back to the original caller, possibly main.
- The recursion step executes while the original call to the function is still “open,” i.e., it has not yet finished executing.
- The recursion step can result in many more such recursive calls, as the function keeps dividing each new subproblem with which the function is called into two conceptual pieces.
- In order for the recursion to eventually terminate, each time the function calls itself with a slightly simpler version of the original problem, this sequence of smaller and smaller problems must eventually converge on the base case.
- At that point, the function recognizes the base case and returns a result to the previous copy of the function, and a sequence of returns ensues up the line until the original call eventually returns the final result to main.

5.20 Recursion (cont.)

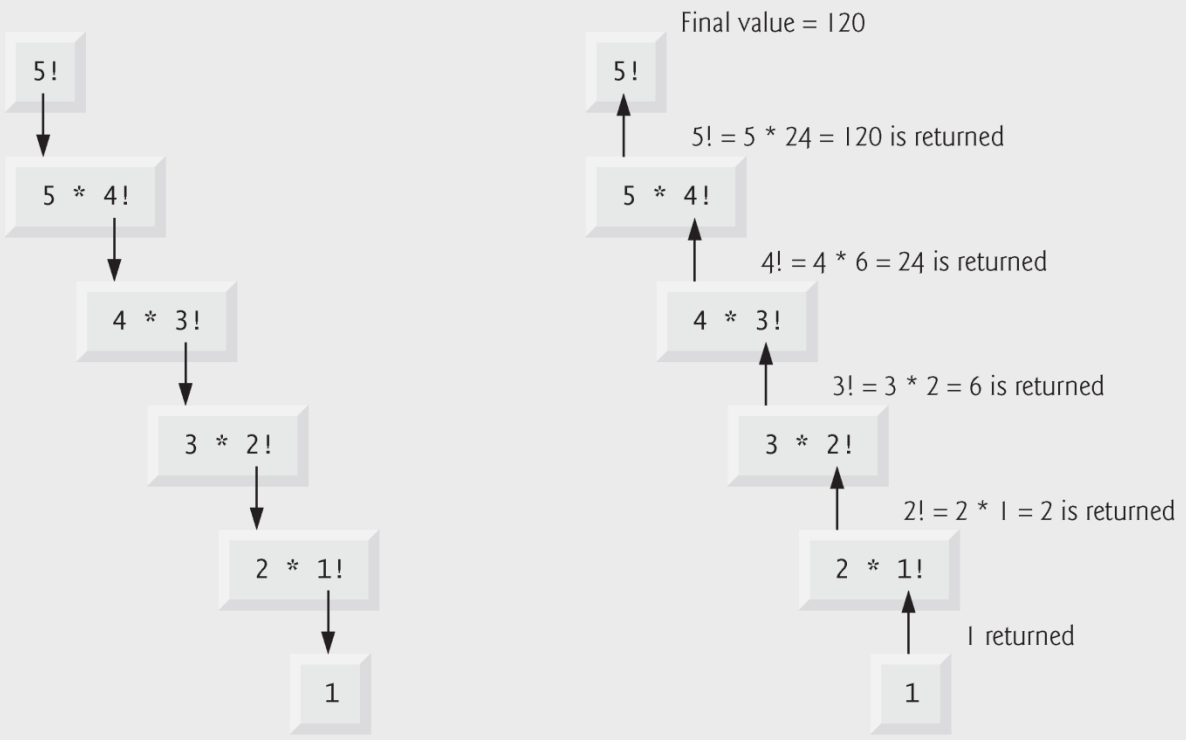
- The factorial of a nonnegative integer n, written n! (and pronounced “n factorial”), is the product
  - n · (n – 1) · (n – 2) · … · 1
- with 1! equal to 1, and 0! defined to be 1.
- For example, 5! is the product 5 · 4 · 3 · 2 · 1, which is equal to 120.
- The factorial of an integer, number, greater than or equal to 0, can be calculated iteratively (nonrecursively) by using a for statement as follows:
  - factorial = 1;
  - for (int counter = number; counter >= 1; counter--)
    factorial *= counter;

5.20 Recursion (cont.)

- A recursive definition of the factorial function is arrived at by observing the following algebraic relationship:
  - $n! = n \cdot (n - 1)!$
- 5! is clearly equal to 5 * 4! as is shown by the following:
  - $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$

  $5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$

  $5! = 5 \cdot (4!)$
- The evaluation of 5! would proceed as shown in Fig. 5.27.
  - Figure 5.27(a) shows how the succession of recursive calls proceeds until 1! is evaluated to be 1, which terminates the recursion.
  - Figure 5.27(b) shows the values returned from each recursive call to its caller until the final value is calculated and returned.



(a) Procession of recursive calls.

(b) Values returned from each recursive call.

Fig. 5.27 | Recursive evaluation of 5!.

5.20 Recursion (cont.)

- Figure 5.28 uses recursion to calculate and print the factorials of the integers 0–10.
- The recursive function factorial (lines 18–24) first determines whether the terminating condition number <= 1 (line 20) is true.
- If number is less than or equal to 1, the factorial function returns 1 (line 21), no further recursion is necessary and the function terminates.
- If number is greater than 1, line 23 expresses the problem as the product of number and a recursive call to factorial evaluating the factorial of number - 1, which is a slightly simpler problem than the original calculation factorial(number).

```
1  // Fig. 5.28: fig05_28.cpp
2  // Demonstrating the recursive function factorial.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  unsigned long factorial( unsigned long ); // function prototype
8
9  int main()
10 {
11    // calculate the factorials of 0 through 10
12    for ( int counter = 0; counter <= 10; counter++ )
13       cout << setw( 2 ) << counter << "! = " << factorial( counter )
14          << endl;
15 } // end main
16
17 // recursive definition of function factorial
18 unsigned long factorial( unsigned long number )
19 {
20    if ( number <= 1 ) // test for base case
21       return 1; // base cases: 0! = 1 and 1! = 1
22    else // recursion step
23       return number * factorial( number - 1 );
24 } // end function factorial
```

Fig. 5.28 | Demonstrating the recursive function `factorial`. (Part 1 of 2.)

```
 0! = 1
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

Fig. 5.28 | Demonstrating the recursive function `factorial`. (Part 2 of 2.)

5.20 Recursion (cont.)

- Function factorial has been declared to receive a parameter of type unsigned long and return a result of type unsigned long.
- This is shorthand notation for unsigned long int.
- The C++ standard requires that a variable of type unsigned long int be at least as big as an int.
- Typically, an unsigned long int is stored in at least four bytes (32 bits); such a variable can hold a value in the range 0 to at least 4294967295.
  - (The data type long int is also stored in at least four bytes and can hold a value at least in the range –2147483648 to 2147483647.)

5.20 Recursion (cont.)

- As can be seen in Fig. 5.28, factorial values become large quickly.
- We chose the data type unsigned long so that the program can calculate factorials greater than 7! on computers with small (such as two-byte) integers.
- Unfortunately, the function factorial produces large values so quickly that even unsigned long does not help us compute many factorial values before even the size of an unsigned long variable is exceeded.
- Variables of type double could be used to calculate factorials of larger numbers.

Common Programming Error 5.21

Either omitting the base case, or writing the recursion step incorrectly so that it does not converge on the base case, causes "infinite" recursion, eventually exhausting memory. This is analogous to the problem of an infinite loop in an iterative (nonrecursive) solution.

5.21 Example Using Recursion: Fibonacci Series

- The Fibonacci series

  0, 1, 1, 2, 3, 5, 8, 13, 21, …

- begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers.
- The series occurs in nature and, in particular, describes a form of spiral.
- The ratio of successive Fibonacci numbers converges on a constant value of 1.618….
  - This number, too, frequently occurs in nature and has been called the golden ratio or the golden mean.
  - Humans tend to find the golden mean aesthetically pleasing.
  - Architects often design windows, rooms and buildings whose length and width are in the ratio of the golden mean.
  - Postcards are often designed with a golden mean length/width ratio.

5.21 Example Using Recursion: Fibonacci Series (cont.)

- The Fibonacci series can be defined recursively as follows:

  fibonacci(0) = 0
  fibonacci(1) = 1
  fibonacci(n) = fibonacci(n – 1) + fibonacci(n – 2)

- The program of Fig. 5.29 calculates the nth Fibonacci number recursively by using function fibonacci.
- Fibonacci numbers tend to become large quickly.
  - We chose the data type unsigned long for the parameter type and the return type in function fibonacci.
- Figure 5.29 shows the execution of the program, which displays the Fibonacci values for several numbers.

```
1  // Fig. 5.29: fig05_29.cpp
2  // Testing the recursive fibonacci function.
3  #include <iostream>
4  using namespace std;
5
6  unsigned long fibonacci( unsigned long ); // function prototype
7
8  int main()
9  {
10    // calculate the fibonacci values of 0 through 10
11    for ( int counter = 0; counter <= 10; counter++ )
12       cout << "fibonacci( " << counter << " ) = "
13          << fibonacci( counter ) << endl;
14
15    // display higher fibonacci values
16    cout << "fibonacci( 20 ) = " << fibonacci( 20 ) << endl;
17    cout << "fibonacci( 30 ) = " << fibonacci( 30 ) << endl;
18    cout << "fibonacci( 35 ) = " << fibonacci( 35 ) << endl;
19 } // end main
20
```

Fig. 5.29 | Demonstrating function `fibonacci`. (Part 1 of 2.)

```
21  // recursive function fibonacci
22  unsigned long fibonacci( unsigned long number )
23  {
24     if ( ( number == 0 ) || ( number == 1 ) ) // base cases
25        return number;
26     else // recursion step
27        return fibonacci( number - 1 ) + fibonacci( number - 2 );
28  } // end function fibonacci
```

```
fibonacci( 0 ) = 0
fibonacci( 1 ) = 1
fibonacci( 2 ) = 1
fibonacci( 3 ) = 2
fibonacci( 4 ) = 3
fibonacci( 5 ) = 5
fibonacci( 6 ) = 8
fibonacci( 7 ) = 13
fibonacci( 8 ) = 21
fibonacci( 9 ) = 34
fibonacci( 10 ) = 55
fibonacci( 20 ) = 6765
fibonacci( 30 ) = 832040
fibonacci( 35 ) = 9227465
```

Fig. 5.29 | Demonstrating function `fibonacci`. (Part 2 of 2.)

5.21 Example Using Recursion: Fibonacci Series (cont.)

- Interestingly, if number is greater than 1, the recursion step (line 27) generates two recursive calls, each for a slightly smaller problem than the original call to fibonacci.
- Figure 5.30 shows how function fibonacci would evaluate fibonacci(3).
- This figure raises some interesting issues about the order in which C++ compilers evaluate the operands of operators.
- This is a separate issue from the order in which operators are applied to their operands, namely, the order dictated by the rules of operator precedence and associativity.
- Figure 5.30 shows that evaluating fibonacci(3) causes two recursive calls, namely, fibonacci(2) and fibonacci(1).

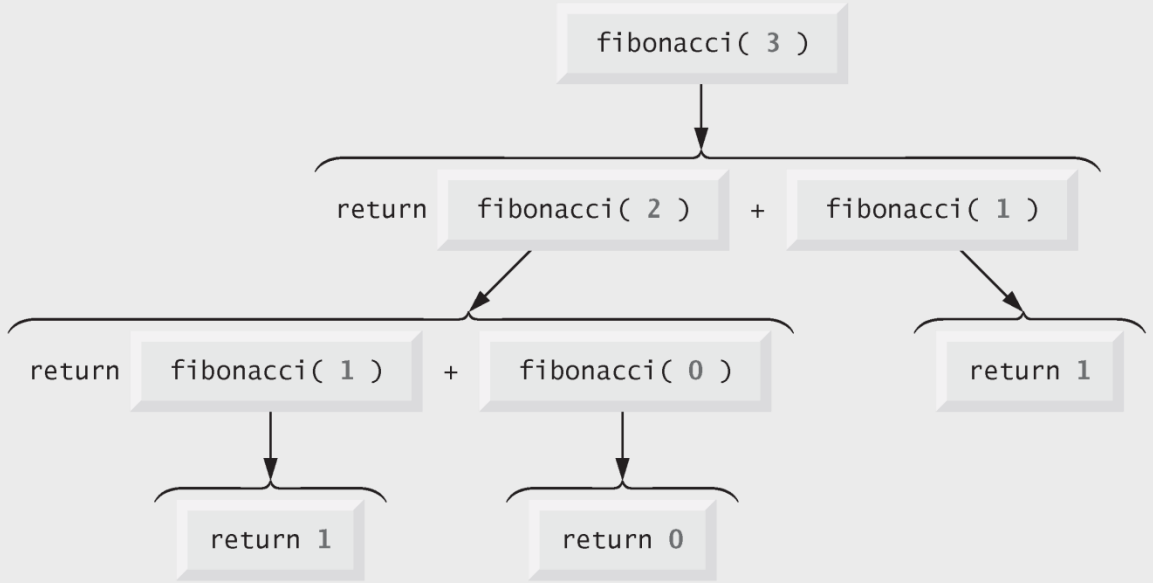


Fig. 5.30 | Set of recursive calls to function `fibonacci`.

5.21 Example Using Recursion: Fibonacci Series (cont.)

- C++ does not specify the order in which the operands of most operators (including +) are to be evaluated.
- Therefore, you must make no assumption about the order in which these calls execute.
- The calls could in fact execute fibonacci(2) first, then fibonacci(1), or they could execute in the reverse order: fibonacci(1), then fibonacci(2).
- In this program and in most others, it turns out that the final result would be the same.
- However, in some programs the evaluation of an operand can have side effects (changes to data values) that could affect the final result of the expression.

5.21 Example Using Recursion: Fibonacci Series (cont.)

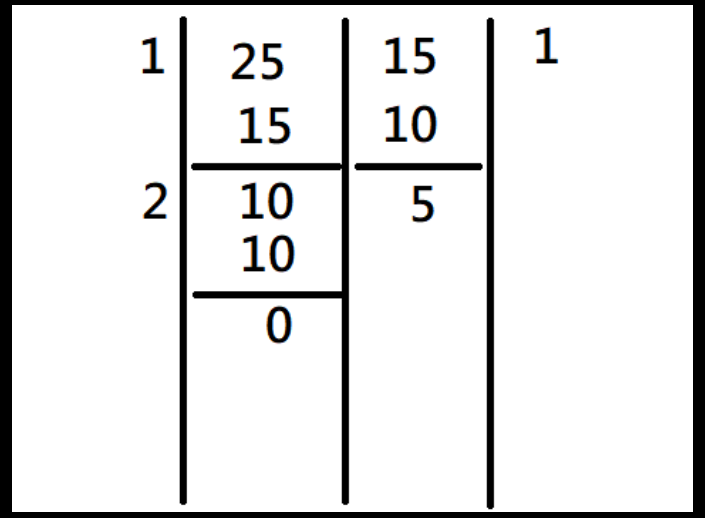
- A word of caution is in order about recursive programs like the one we use here to generate Fibonacci numbers.
- Each level of recursion in function fibonacci has a doubling effect on the number of function calls; i.e., the number of recursive calls that are required to calculate the nth Fibonacci number is on the order of 2^n.
- Calculating only the 20th Fibonacci number would require on the order of 2^20 or about a million calls, calculating the 30th Fibonacci number would require on the order of 2^30 or about a billion calls, and so on.
- Computer scientists refer to this as exponential complexity.

Performance Tip 5.8

Avoid Fibonacci-style recursive programs that result in an exponential "explosion" of calls.

29

In-Class Exercise



- Euclidean algorithm (輾轉相除法)

- Please write a function int GCD(int x, int y), which return the greatest common divisor of x and y by recursive.

- Hint:
  - 1. use mod(%)
  - 2. The base case is x%y==0
- cout<< GCD(6497,3869) << endl;

5.22 Recursion vs. Iteration

- In the two previous sections, we studied two functions that easily can be implemented recursively or iteratively.
- This section compares the two approaches and discusses why you might choose one approach over the other in a particular situation.

5.22 Recursion vs. Iteration (cont.)

- Both iteration and recursion are based on a control statement: Iteration uses a repetition structure; recursion uses a selection structure.
- Both iteration and recursion involve repetition: Iteration explicitly uses a repetition structure; recursion achieves repetition through repeated function calls.

5.22 Recursion vs. Iteration (cont.)

- Iteration and recursion both involve a termination test: Iteration terminates when the loop-continuation condition fails; recursion terminates when a base case is recognized.
- Iteration with counter-controlled repetition and recursion both gradually approach termination: Iteration modifies a counter until the counter assumes a value that makes the loop-continuation condition fail; recursion produces simpler versions of the original problem until the base case is reached.

5.22 Recursion vs. Iteration (cont.)

- Both iteration and recursion can occur infinitely: An infinite loop occurs with iteration if the loop-continuation test never becomes false; infinite recursion occurs if the recursion step does not reduce the problem during each recursive call in a manner that converges on the base case.

5.22 Recursion vs. Iteration (cont.)

- To illustrate the differences between iteration and recursion, let's examine an iterative solution to the factorial problem (Fig. 5.31).
- A repetition statement is used (lines 23–24 of Fig. 5.31) rather than the selection statement of the recursive solution (lines 20–23 of Fig. 5.28).
- Both solutions use a termination test.
- In the recursive solution, line 20 tests for the base case.
- In the iterative solution, line 23 tests the loop-continuation condition—if the test fails, the loop terminates.
- Finally, instead of producing simpler versions of the original problem, the iterative solution uses a counter that's modified until the loop-continuation condition becomes false.

```
1  // Fig. 5.31: fig05_31.cpp
2  // Testing the iterative factorial function.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  unsigned long factorial( unsigned long ); // function prototype
8
9  int main()
10 {
11    // calculate the factorials of 0 through 10
12    for ( int counter = 0; counter <= 10; counter++ )
13       cout << setw( 2 ) << counter << "! = " << factorial( counter )
14          << endl;
15 } // end main
16
```

Fig. 5.31 | Iterative factorial solution. (Part 1 of 2.)

```
17  // iterative function factorial
18  unsigned long factorial( unsigned long number )
19  {
20     unsigned long result = 1;
21
22     // iterative factorial calculation
23     for ( unsigned long i = number; i >= 1; i-- )
24        result *= i;
25
26     return result;
27  } // end function factorial
```

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

Fig. 5.31 | Iterative factorial solution. (Part 2 of 2.)

5.22 Recursion vs. Iteration (cont.)

- Recursion has many negatives.
- It repeatedly invokes the mechanism, and consequently the overhead of function calls.
- This can be expensive in both processor time and memory space.
- Each recursive call causes another copy of the function (actually only the function's variables) to be created; this can consume considerable memory.
- Iteration normally occurs within a function, so the overhead of repeated function calls and extra memory assignment is omitted.

Software Engineering Observation 5.15

Any problem that can be solved recursively can also be solved iteratively (nonrecursively). A recursive approach is normally chosen when the recursive approach more naturally mirrors the problem and results in a program that's easier to understand and debug. Another reason to choose a recursive solution is that an iterative solution is not apparent.

Performance Tip 5.9

Avoid using recursion in performance situations. Recursive calls take time and consume additional memory.

Common Programming Error 5.23

Accidentally having a nonrecursive function call itself, either directly or indirectly (through another function), is a logic error.

5.22 Recursion vs. Iteration (cont.)

- Figure 5.32 summarizes the recursion examples and exercises in the text.

Location in text	Recursion examples and exercises
Chapter 5	
Section 5.20, Fig. 5.28	Factorial function
Section 5.21, Fig. 5.29	Fibonacci function
Exercise 5.36	Raising an integer to an integer power
Exercise 5.38	Towers of Hanoi
Exercise 5.40	Visualizing recursion
Exercise 5.41	Greatest common divisor
Exercise 5.45, Exercise 5.46	Mystery "What does this program do?" exercise
Chapter 6	
Exercise 6.18	Mystery "What does this program do?" exercise
Exercise 6.21	Mystery "What does this program do?" exercise
Exercise 6.31	Selection sort
Exercise 6.32	Determine whether a string is a palindrome
Exercise 6.33	Linear search
Exercise 6.34	Eight Queens
Exercise 6.35	Print an array

Fig. 5.32 | Summary of recursion examples and exercises in the text. (Part 1 of 3.)

Location in text	Recursion examples and exercises
Exercise 6.36	Print a string backward
Exercise 6.37	Minimum value in an array
Chapter 7	
Exercise 7.15	Quicksort
Exercise 7.16	Maze traversal
Exercise 7.17	Generating mazes randomly
Chapter 19	
Section 19.3.3, Figs. 19.5–19.7	Mergesort
Exercise 19.8	Linear search
Exercise 19.9	Binary search
Exercise 19.10	Quicksort
Chapter 20	
Section 20.7, Figs. 20.20–20.22	Binary tree insert
Section 20.7, Figs. 20.20–20.22	Preorder traversal of a binary tree
Section 20.7, Figs. 20.20–20.22	Inorder traversal of a binary tree

Fig. 5.32 | Summary of recursion examples and exercises in the text. (Part 2 of 3.)

Location in text	Recursion examples and exercises
Section 20.7, Figs. 20.20–20.22	Postorder traversal of a binary tree
Exercise 20.20	Print a linked list backward
Exercise 20.21	Search a linked list
Exercise 20.22	Binary tree delete
Exercise 20.23	Binary tree search
Exercise 20.24	Level order traversal of a binary tree
Exercise 20.25	Printing tree

Fig. 5.32 | Summary of recursion examples and exercises in the text. (Part 3 of 3.)